

Views, Types and Viewtypes*

Hongwei Xi and Dengping Zhu

Computer Science Department

Boston University

October 23, 2004

Abstract

The need for direct memory manipulation through pointers is essential in many applications. However, it is also commonly understood that the use (or probably misuse) of pointers is a rich source for program errors. In this paper, we design and then formalize a type system that can effectively prevent dangling pointers from being ever accessed during the evaluation of a well-type program. In particular, we present an approach that makes novel use of linear types in support of the construction of memory-safe programs while allowing explicit pointer manipulation such as pointer arithmetic.

Keywords: View, Type, Viewtype, Applied Type System, ATS

*Partially supported by NSF grant no. CCR-0229480

```

dataview arrayView (type, int, addr) =
  | {a:type, l:addr} ArrayNone (a, 0, l)
  | {a:type, n:int, l:addr | n >= 0}
    ArraySome (a, n+1, l) of (a @ l, arrayView (a, n, l+1))

```

Figure 1: An example of recursive stateful view

1 Introduction

The need for direct memory manipulation through pointers is essential in many applications and especially in those that involve systems programming. However, it is also commonly understood that the use (or probably misuse) of pointers is often a rich source for program errors. Therefore, a type system is highly sought after that can guarantee memory safety while allowing flexible use of pointers in programming.

We have recently presented a type system *ATS/SV* (Xi et al., 2004) that make use of a notion called *stateful views* to model memory layouts. For instance, given a type T and an address L , we can form a (primitive) stateful view $T@L$ to mean that a value of type T is stored at the address L . We can also form new stateful views in terms of primitive stateful views. For instance, given types T_1 and T_2 and an address L , we can form a view $(T_1@L) \otimes (T_2@L + 1)$ to mean that a value of type T_1 and another value of type T_2 are stored at addresses L and $L + 1$, respectively, where $L + 1$ stands for the address immediately after L . Intuitively, a view is like a type, but it is linear. Given a term of some view V , we often say that the term proves the view V and thus refer to the term as a *proof* (of V).

In order to model more sophisticated memory layouts, we need to form recursive stateful views. For instance, we may use the concrete syntax in Figure 1 to declare a (dependent) view constructor *arrayView*: Given a type T , an integer I and an address L , *arrayView*(T, I, L) forms a view basically stating that there are I values of type T stored at addresses $L, L + 1, \dots, L + I - 1$. There are two proof constructors *ArrayNone* and *ArraySome* associated with *arrayView*; *ArrayNone* is a proof of *arrayView*($T, 0, L$) for any type T and address L ; *ArraySome*(pf_1, pf_2) is a proof of *arrayView*($T, I + 1, L$) for any type T , integer I and address L if pf_1 and pf_2 are proofs of views $T@L$ and *arrayView*($T, I, L + 1$), respectively. Later, we will present some simple examples to illustrate how programming with views can actually be done.

What is missing in *ATS/SV* (Xi et al., 2004) is the concept of *viewtype*, which we now introduce. Given a view V and a type T , we can form a *viewtype* $V \wedge T$ such that a value of the type $V \wedge T$ is a pair $pf \wedge v$ in which pf is a proof of V and v is a value of type T . For instance, the following type can be assigned to a function *read_L* that reads from the address L :

$$(T@L) \wedge \mathbf{ptr}(L) \rightarrow (T@L) \wedge T$$

Note that $\mathbf{ptr}(L)$ is the singleton type for the only pointer pointing to the address L . When applied to a value $pf_1 \wedge L$ of type $(T@L) \wedge \mathbf{ptr}(L)$, the function *read_L* returns a value $pf_2 \wedge v$, where v is the value of type T that is supposed to be stored at L . Both pf_1 and pf_2 are proofs of the view

$T@L$, and we may think that the call to $read_L$ consumes pf_1 and then produces pf_2 . Similarly, the following type can be assigned to a function $write_L$ that writes a value of type T_2 to the address L where a value of type T_1 is stored:

$$(T_1@L) \wedge (\mathbf{ptr}(L) * T_2) \rightarrow (T_2@L) \wedge \mathbf{1}$$

Note that $\mathbf{1}$ stands for the unit type. Of course, once we allow universal quantification over types and addresses, we can then assign the read and write functions the following types:

$$\begin{aligned} read & : \quad \forall \tau. \forall \lambda. (\tau@L) \wedge \mathbf{ptr}(L) \rightarrow (\tau@L) \wedge \tau \\ write & : \quad \forall \tau_1. \forall \tau_2. \forall \lambda. (\tau_1@L) \wedge (\mathbf{ptr}(L) * \tau_2) \rightarrow (\tau_2@L) \wedge \mathbf{1} \end{aligned}$$

where we use τ and λ to range over types and addresses, respectively.

In *ATS/SV* (Xi et al., 2004), we have already presented a variety of examples that attest to the expressiveness of views in capturing program invariants on memory layouts. However, the concept of viewtype is hidden in *ATS/SV* (where it is called *computation type*), which makes the formulation of *ATS/SV* rather *ad hoc* in nature. The primary contribution of the paper precisely lies in the recognition of this important concept of viewtype and then the formalization of a type system in support of this concept.

We organize the rest of the paper as follows. In Section 2, we formalize a language λ_{view} in which views, types and viewtypes are all supported. We then briefly mention in Section 3 an extension $\lambda_{view}^{\forall, \exists}$ of λ_{view} in which we support dependent types as well as polymorphic types. In Section 4, we present some examples to show how views can be used in practical programming. Lastly, we mention some related work and conclude.

2 Formal Development

In this section, we formally present a language λ_{view} in which the type system supports views, types and viewtypes. The main purpose of formalizing λ_{view} is to allow for a gentle introduction to unfamiliar concepts such as view and viewtype. To some extent, λ_{view} can be compared to the simply type lambda-calculus, which forms the core of more advanced typed lambda-calculi. We will later extend λ_{view} to $\lambda_{view}^{\forall, \exists}$ with dependent types as well as polymorphic types, greatly facilitating the use of views and viewtypes in programming.

The syntax of λ_{view} is given in Figure 2. We use V for views and L for addresses. We use $\mathbf{l}_0, \mathbf{l}_1, \dots$ for infinitely many distinct constant addresses, which one may assume to be represented as natural numbers. Also, we write l for a constant address. We use \underline{x} for proof variables and \underline{t} for proof terms. For each address l , \underline{l} is a constant proof term, whose meaning is to become clear soon. We use Π for a proof variable context, which assigns views to proof variables.

We use T and VT for types and viewtypes, respectively. Note that a type T is just a special form of viewtype. We use t for dynamic terms (that is, programs) and v for values. We write Δ^i (Δ^l) for an intuitionistic (a linear) dynamic variable context, which assign types (viewtypes) to dynamic variables, and Δ for a (combined) dynamic context of the form $(\Delta^i; \Delta^l)$. Given $\Delta = (\Delta^i; \Delta^l)$, we

addresses	$L ::= \mathbf{l}_0 \mid \mathbf{l}_1 \mid \dots$
views	$V ::= T @ L \mid V_1 \otimes V_2 \mid V_1 \multimap V_2$
proof terms	$\underline{t} ::= x \mid \underline{l} \mid \langle \underline{t}_1, \underline{t}_2 \rangle \mid \mathbf{let} \langle \underline{x}_1, \underline{x}_2 \rangle = \underline{t}_1 \mathbf{in} \underline{t}_2 \mid \lambda \underline{x}. \underline{t} \mid \underline{t}_1(\underline{t}_2)$
proof var. ctx.	$\Pi ::= \emptyset \mid \Pi, a : V$
types	$T ::= \mathbf{Bool} \mid \mathbf{Int} \mid \mathbf{ptr}(L) \mid \mathbf{1} \mid V \supset VT \mid T * T \mid VT \rightarrow VT$
viewtypes	$VT ::= \mathbf{Bool} \mid \mathbf{Int} \mid \mathbf{ptr}(L) \mid \mathbf{1} \mid V \wedge VT \mid V \supset_0 VT \mid V \supset VT \mid VT_1 * VT_2 \mid VT \rightarrow_0 VT \mid VT \rightarrow VT$
dyn. terms	$t ::= x \mid f \mid cc(t_1, \dots, t_n) \mid cf(t_1, \dots, t_n) \mid read(\underline{t}, t) \mid write(\underline{t}, t_1, t_2) \mid \mathbf{if}(t_1, t_2, t_3) \mid read(\underline{t}, t) \mid write(\underline{t}, t_1, t_2) \mid \underline{t} \wedge t \mid \mathbf{let} \underline{x} \wedge x = t_1 \mathbf{in} t_2 \mid \lambda \underline{x}. v \mid t(\underline{t}) \mid \langle \rangle \mid \langle t_1, t_2 \rangle \mid \mathbf{let} \langle x_1, x_2 \rangle = t_1 \mathbf{in} t_2 \mid \mathbf{lam} x.t \mid \mathbf{app}(t_1, t_2) \mid \mathbf{fix} f.t$
values	$v ::= x \mid cc(v_1, \dots, v_n) \mid \underline{t} \wedge v \mid \lambda \underline{x}. v \mid \langle v_1, v_2 \rangle \mid \mathbf{lam} x.d$
int. dyn. var. ctx.	$\Delta^i ::= \emptyset \mid \Delta^i, x : T$
lin. dyn. var. ctx.	$\Delta^l ::= \emptyset \mid \Delta^l, x : VT$
dyn. var. ctx.	$\Delta ::= (\Delta^i; \Delta^l)$
state types	$\mu ::= [] \mid \mu[l \mapsto T]$
states	$ST ::= [] \mid ST[l \mapsto v]$

Figure 2: The syntax for λ_{view}

$$\begin{array}{c}
 \frac{}{\emptyset \vdash_{[\mu \mapsto T]} \underline{l} : T@l} \text{ (vw-addr)} \\
 \frac{}{\emptyset, \underline{x} : V \vdash_{\square} \underline{x} : V} \text{ (vw-var)} \\
 \frac{\Pi_1 \vdash_{\mu_1} \underline{t}_1 : V_1 \quad \Pi_2 \vdash_{\mu_2} \underline{t}_2 : V_2}{\Pi_1, \Pi_2 \vdash_{\mu_1 \otimes \mu_2} \langle \underline{t}_1, \underline{t}_2 \rangle : V_1 \otimes V_2} \text{ (vw-tup)} \\
 \frac{\Pi_1 \vdash_{\mu_1} \underline{t}_1 : V_1 \otimes V_2 \quad \Pi_2, \underline{x}_1 : V_1, \underline{x}_2 : V_2 \vdash_{\mu_2} \underline{t}_2 : V}{\Pi_1, \Pi_2 \vdash_{\mu_1 \otimes \mu_2} \mathbf{let} \langle \underline{x}_1, \underline{x}_2 \rangle = \underline{t}_1 \mathbf{in} \underline{t}_2 : V} \text{ (vw-let)} \\
 \frac{\Pi, \underline{x} : V_1 \vdash_{\mu} \underline{t} : V_2}{\Pi \vdash_{\mu} \lambda \underline{x}. \underline{t} : V_1 \multimap V_2} \text{ (vw-lam)} \\
 \frac{\Pi_1 \vdash_{\mu_1} \underline{t}_1 : V_1 \multimap V_2 \quad \Pi_2 \vdash_{\mu_2} \underline{t}_2 : V_1}{\Pi_1, \Pi_2 \vdash_{\mu_1 \otimes \mu_2} \underline{t}_1(\underline{t}_2) : V_2} \text{ (vw-app)}
 \end{array}$$

Figure 3: The rules for assigning views to proof terms

may use $\Delta, x : VT$ for $(\Delta^i; \Delta^l, x : VT)$; in case the viewtype VT is actually a type, we may also use $\Delta, x : VT$ for $(\Delta^i, x : VT; \Delta^l)$. In addition, given $\Delta_1 = (\Delta^i; \Delta_1^l)$ and $\Delta_2 = (\Delta^i; \Delta_2^l)$, we write $\Delta_1 \uplus \Delta_2$ for $(\Delta^i; \Delta_1^l, \Delta_2^l)$.

We use x and f for dynamic **lam**-variables and **fix**-variables, respectively, and xf for either x or f ; a **lam**-variable is a value while a **fix**-variable is not. We use c for a dynamic constant, which is either a function cf or a constructor cc . Each constant is given a constant type (or c-type) of the form $(T_1, \dots, T_n) \Rightarrow T$, where n is the arity of c . We may write cc for $cc()$. For instance, each address l is given the c-type $() \Rightarrow \mathbf{ptr}(l)$; each boolean value is given the c-type $() \Rightarrow \mathbf{Bool}$; each integer is given the c-type $() \Rightarrow \mathbf{Int}$; the equality function on integers can be given the c-type: $(\mathbf{Int}, \mathbf{Int}) \Rightarrow \mathbf{Bool}$.

We use μ and ST for state types and states, respectively. A state type maps constant addresses to types while a state maps constant addresses to values. We use \square for the empty mapping and $\mu[l \mapsto T]$ for the mapping that extends μ with a link from l to T . It is implicitly assumed that l is not in the domain $\mathbf{dom}(\mu)$ of μ when $\mu[l \mapsto T]$ is formed. Given two state types μ_1 and μ_2 with disjoint domains, we write $\mu_1 \otimes \mu_2$ for the standard union of μ_1 and μ_2 . Similar notations are also applicable to states.

The rules for assigning views to proofs are given in Figure 3. So far only logic constructs in the multiplicative fragment of intuitionistic linear logic are involved in forming views, and we plan to handle logic constructs in the additive fragment of intuitionistic linear logic in future. A judgment of the form $\Pi \vdash_{\mu} \underline{t} : V$ means that \underline{t} can be assigned the view V if the variables and constants in \underline{t} are assigned views according to Π and μ , respectively.

The rules for assigning viewtypes (which include types) are given in Figure 4. We use $\supset_?$ for \supset or \supset_0 in the rule (**ty-vapp**) and $\rightarrow_?$ for \rightarrow or \rightarrow_0 in the rule (**ty-app**). Intuitively, a type of the form $V \supset_? VT$ is for functions from proofs of view V to values of viewtype VT . Similarly, a type of the form $VT_1 \rightarrow_? VT_2$ is for functions from values of viewtype VT_1 to values of viewtype VT_2 .

$$\begin{array}{c}
 \overline{\emptyset; (\Delta^i; \emptyset), x : VT \vdash_{\square} x : VT} \quad \text{(ty-var)} \\
 \frac{\vdash c : (T_1, \dots, T_n) \Rightarrow T \quad \Pi_i; \Delta_i \vdash_{\mu_i} t_i : T_i \text{ for } 1 \leq i \leq n}{\Pi_1, \dots, \Pi_n; \Delta_1 \uplus \dots \uplus \Delta_n \vdash_{\mu_1 \otimes \dots \otimes \mu_n} c(t_1, \dots, t_n) : T} \quad \text{(ty-cst)} \\
 \frac{\Pi_1; \Delta_1 \vdash_{\mu_1} t_1 : \mathbf{Bool} \quad \Pi_2; \Delta_2 \vdash_{\mu_2} t_2 : VT \quad \Pi_2; \Delta_2 \vdash_{\mu_2} t_3 : VT}{\Pi_1, \Pi_2; \Delta_1 \uplus \Delta_2 \vdash_{\mu_1 \otimes \mu_2} \mathbf{if}(t_1, t_2, t_3) : VT} \quad \text{(ty-if)} \\
 \frac{\Pi_1 \vdash_{\mu_1} \underline{t} : V \quad \Pi_2; \Delta \vdash_{\mu_2} t : VT}{\Pi_1, \Pi_2; \Delta \vdash_{\mu_1 \otimes \mu_2} \langle \underline{t}, t \rangle : V \wedge VT} \quad \text{(ty-vtup)} \\
 \frac{\Pi_1; \Delta_1 \vdash_{\mu_1} t_1 : V \wedge VT_1 \quad \Pi_2, \underline{x} : V; \Delta_2, x : VT_1 \vdash_{\mu_2} t_2 : VT_2}{\Pi_1, \Pi_2; \Delta_1 \uplus \Delta_2 \vdash_{\mu_1 \otimes \mu_2} \mathbf{let } \underline{x} \wedge x = t_1 \mathbf{ in } t_2 : VT_2} \quad \text{(ty-vlet)} \\
 \overline{\emptyset; (\Delta^i; \emptyset) \vdash_{\square} \langle \rangle : \mathbf{1}} \quad \text{(ty-unit)} \\
 \frac{\Pi_1; \Delta_1 \vdash_{\mu_1} t_1 : VT_1 \quad \Pi_2; \Delta_2 \vdash_{\mu_2} t_2 : VT_2}{\Pi_1, \Pi_2; \Delta_1 \uplus \Delta_2 \vdash_{\mu_1 \otimes \mu_2} \langle t_1, t_2 \rangle : VT_1 * VT_2} \quad \text{(ty-tup)} \\
 \frac{\Pi_1; \Delta_1 \vdash_{\mu_1} t_1 : VT_1 * VT_2 \quad \Pi_2; \Delta_2, x_1 : VT_1, x_2 : VT_2 \vdash_{\mu_2} t_2 : VT}{\Pi_1, \Pi_2; \Delta_1 \uplus \Delta_2 \vdash_{\mu_1 \otimes \mu_2} \mathbf{let } \langle x_1, x_2 \rangle = t_1 \mathbf{ in } t_2 : VT} \quad \text{(ty-let)} \\
 \frac{\Pi, \underline{x} : V; \Delta \vdash_{\mu} v : VT}{\Pi; \Delta \vdash_{\mu} \lambda \underline{x}. v : V \supset_0 VT} \quad \text{(ty-vlam0)} \\
 \frac{\emptyset, \underline{x} : V; (\Delta^i; \emptyset) \vdash_{\square} v : VT}{\emptyset; (\Delta^i; \emptyset) \vdash_{\square} \lambda \underline{x}. v : V \supset VT} \quad \text{(ty-vlam)} \\
 \frac{\Pi_1; \Delta \vdash_{\mu_1} t : V \supset_? VT \quad \Pi_2 \vdash_{\mu_2} \underline{t} : V}{\Pi_1, \Pi_2; \Delta \vdash_{\mu_1 \otimes \mu_2} t(\underline{t}) : VT} \quad \text{(ty-vapp)} \\
 \frac{\Pi; \Delta, x : VT_1 \vdash_{\mu} t : VT_2}{\Pi; \Delta \vdash_{\mu} \mathbf{lam } x.t : VT_1 \rightarrow_0 VT_2} \quad \text{(ty-lam0)} \\
 \frac{\emptyset; (\Delta^i; \emptyset), x : VT_1 \vdash_{\square} t : VT_2}{\emptyset; (\Delta^i; \emptyset) \vdash_{\square} \mathbf{lam } x.t : VT_1 \rightarrow VT_2} \quad \text{(ty-lam)} \\
 \frac{\Pi_1; \Delta_1 \vdash_{\mu_1} t_1 : VT_1 \rightarrow_? VT_2 \quad \Pi_2; \Delta_2 \vdash_{\mu_2} t_2 : VT_1}{\Pi_1, \Pi_2; \Delta_1 \uplus \Delta_2 \vdash_{\mu_1 \otimes \mu_2} \mathbf{app}(t_1, t_2) : VT_2} \quad \text{(ty-app)} \\
 \frac{\emptyset; (\Delta^i, f : T; \emptyset) \vdash_{\square} t : T}{\emptyset; (\Delta^i; \emptyset) \vdash_{\square} \mathbf{fix } f.t : T} \quad \text{(ty-fix)} \\
 \frac{\Pi_1 \vdash_{\mu_1} \underline{t} : T @ L \quad \Pi_2; \Delta \vdash_{\mu_2} t : \mathbf{ptr}(L)}{\Pi_1, \Pi_2; \Delta \vdash_{\mu_1 \otimes \mu_2} \mathbf{read}(\underline{t}, t) : (T @ L) \wedge T} \quad \text{(ty-read)} \\
 \frac{\Pi_1 \vdash_{\mu_1} \underline{t} : T @ L \quad \Pi_2; \Delta \vdash_{\mu_2} t_1 : \mathbf{ptr}(L) \quad \Pi_3 \vdash_{\mu_3} t_2 : T'}{\Pi_1, \Pi_2, \Pi_3; \Delta \vdash_{\mu_1 \otimes \mu_2 \otimes \mu_3} \mathbf{write}(\underline{t}, t_1, t_2) : (T' @ L) \wedge \mathbf{1}} \quad \text{(ty-write)}
 \end{array}$$

Figure 4: The rules for assigning viewtypes to dynamic terms

Proposition 2.1 Assume that $\emptyset; (\emptyset; \emptyset) \vdash_{\mu} v : T$ is derivable. Then the state type μ must equal $\llbracket \cdot \rrbracket$.

Proof By a careful inspection of the rules in Figure 4. ■

Proposition 2.1 is of great importance. Intuitively, the proposition states that if a closed value is assigned a type T , then the value can be constructed without consuming resources (in the sense of proof constants $\llbracket \cdot \rrbracket$) and thus is allowed to be duplicated. For instance, a value of type $V \supset VT$ may be duplicated while a value of type $V \supset_0 VT$ may not. The difference between $VT_1 \rightarrow VT_2$ and $VT_1 \rightarrow_0 VT_2$ is similar.

We use a judgment of the form $ST \models V$ to mean that the state ST entails the view V . The rules for deriving such judgments are given below:

$$\frac{\emptyset; (\emptyset; \emptyset) \vdash_{\llbracket \cdot \rrbracket} v : T}{\llbracket l \mapsto v \rrbracket \models T@l} \quad \frac{ST_1 \models V_1 \quad ST_2 \models V_2}{ST_1 \otimes ST_2 \models V_1 \otimes V_2} \quad \frac{ST_0 \otimes ST \models V_2 \text{ for each } ST_0 \models V_1}{ST \models V_1 \multimap V_2}$$

Lemma 2.2 Assume $\Pi \vdash_{\mu} \underline{t} : V$ is derivable for $\Pi = \underline{x}_1 : V_1, \dots, \underline{x}_n : V_n$. If $ST = ST_0 \otimes ST_1 \otimes \dots \otimes ST_n$ and $ST_0 : \mu$ and $ST_i \models V_i$ for $1 \leq n$, then $ST \models V$ holds.

Proof By structural induction on the derivation \mathcal{D} of $\Pi \vdash_{\mu} \underline{t} : V$.

- Assume \mathcal{D} is of the following form:

$$\frac{\Pi_1 \vdash_{\mu_1} \underline{t}_1 : V_1 \quad \Pi_2 \vdash_{\mu_2} \underline{t}_2 : V_2}{\Pi_1, \Pi_2 \vdash_{\mu_1 \otimes \mu_2} \langle \underline{t}_1, \underline{t}_2 \rangle : V_1 \otimes V_2} \text{ (vw-tup)}$$

where $\Pi_1 = x_1, \dots, x_i$ and $\Pi_2 = x_{i+1}, \dots, x_n$ for some $1 \leq i \leq n$, and $\mu = \mu_1 \otimes \mu_2$, and $V = V_1 \otimes V_2$. Thus, $ST_0 = ST_{01} \otimes ST_{02}$ such that both $ST_{01} : \mu_1$ and $ST_{02} : \mu_2$ hold. By induction hypothesis, we have $ST_{01} \otimes ST_1 \otimes \dots \otimes ST_i \models V_1$ and $ST_{02} \otimes ST_{i+1} \otimes \dots \otimes ST_n \models V_2$. Hence, $ST \models V_1 \otimes V_2$.

- Assume \mathcal{D} is of the following form

$$\frac{\Pi, \underline{x} : V_1 \vdash_{\mu} \underline{t} : V_2}{\Pi \vdash_{\mu} \lambda \underline{x}. \underline{t} : V_1 \multimap V_2} \text{ (vw-lam)}$$

where $V = V_1 \multimap V_2$. Assume $ST' \models V_1$ for some ST' disjoint from ST . By induction hypothesis, $ST \otimes ST' \models V_2$ holds. By definition, $ST \models V_1 \multimap V_2$ holds.

The other cases can be handled similarly. ■

We use $[\underline{x}_1, \dots, \underline{x}_n \mapsto \underline{t}_1, \dots, \underline{t}_n]$ for a substitution that maps \underline{x}_i to \underline{t}_i for $1 \leq i \leq n$. Similarly, we use $[xf_1, \dots, xf_n \mapsto t_1, \dots, t_n]$ for a substitution that maps xf_i to t_i for $1 \leq i \leq n$.

Definition 2.3 We define redexes as follows:

1. let $\langle \underline{x}, x \rangle = \langle \underline{t}, v \rangle$ in t is a redex, and its reduction is $t[\underline{x} \mapsto \underline{t}][x \mapsto v]$.
2. $(\lambda \underline{x}. v)(\underline{t})$ is a redex, and its reduction is $v[\underline{x} \mapsto \underline{t}]$.

3. **let** $\langle x_1, x_2 \rangle = \langle v_1, v_2 \rangle$ **in** t is a redex, and its reduction is $t[x_1, x_2 \mapsto v_1, v_2]$.
4. **app(lam** $x.t, v)$ is a redex, and its reduction is $t[x \mapsto v]$.
5. **fix** $f.t$ is a redex, and its reduction is $t[f \mapsto \mathbf{fix} f.t]$.

We use E for evaluation contexts, which are defined as follows:

$$\begin{aligned} \text{evaluation context } E ::= & \quad [] \mid c(v_1, \dots, v_{i-1}, E, t_{i+1}, \dots, t_n) \mid \\ & \quad \text{read}(\underline{t}, E) \mid \text{write}(\underline{t}, E, t) \mid \text{write}(\underline{t}, v, E) \mid \\ & \quad \underline{t} \wedge E \mid \mathbf{let} \underline{x} \wedge x = E \mathbf{in} t \mid E(\underline{t}) \mid \\ & \quad \langle E, t \rangle \mid \langle v, E \rangle \mid \mathbf{let} \langle x_1, x_2 \rangle = E \mathbf{in} t \mid \\ & \quad \mathbf{app}(E, t) \mid \mathbf{app}(v, E) \end{aligned}$$

Given E and t , we write $E[t]$ for the dynamic term obtained from replacing the hole $[]$ in E with t . Note that such a replacement can never cause a free variable to be captured. Given ST_1, ST_2 and t_1, t_2 , we write $(ST_1, t_1) \rightarrow_{ev/st} (ST_2, t_2)$ if

1. $t_1 = E[t]$ and $t_2 = E[t']$ for some redex t and its reduction, or
2. $t_1 = E[\text{read}(\underline{t}, l)]$ for some $l \in \mathbf{dom}(ST_1)$ and $t_2 = E[\langle \underline{t}, ST_1(l) \rangle]$ and $ST_2 = ST_1$, or
3. $t_1 = E[\text{write}(\underline{t}, l, v)]$ for some $l \in \mathbf{dom}(ST_1)$ and $t_2 = E[\langle \underline{t}, \langle \rangle \rangle]$ and $ST_2 = ST_1[l := v]$.

We write $ST[l := v]$ for a state that maps l to v and coincides with ST elsewhere. Note that we implicitly assume $l \in \mathbf{dom}(ST)$ when writing $ST[l := v]$.

Lemma 2.4 (Substitution) *We have the following:*

1. Assume that both $\Pi_1 \vdash_{\mu_1} \underline{t}_1 : V_1$ and $\Pi_2, \underline{x} : V_1 \vdash_{\mu_2} \underline{t}_2 : V_2$ are derivable. Then $\Pi_2, \underline{x} : V_1 \vdash_{\mu_1 \otimes \mu_2} \underline{t}_2[\underline{x} \mapsto \underline{t}_1] : V_2$ is also derivable.
2. Assume that both $\Pi_1 \vdash_{\mu_1} \underline{t} : V$ and $\Pi_2, \underline{x} : V; \Delta \vdash_{\mu_2} t : VT$ are derivable. Then $\Pi_1, \Pi_2; \Delta \vdash_{\mu_1 \otimes \mu_2} t[\underline{x} \mapsto \underline{t}] : VT$ is also derivable.
3. Assume that both $\emptyset; (\emptyset; \emptyset) \vdash_{\mu_1} v : VT_1$ and $\Pi; \Delta, x : VT_1 \vdash_{\mu_2} t : VT_2$ are derivable. Then $\Pi; \Delta \vdash_{\mu_1 \otimes \mu_2} t[x \mapsto v] : VT_2$ is also derivable.

Proof By standard structural induction. In particular, we encounter a need for Proposition 2.1 when proving (3). We now show a few cases in the proof of (3), which proceeds by structural induction on the derivation \mathcal{D} of $\Pi; \Delta, x : VT_1 \vdash_{\mu_2} t : VT_2$.

- \mathcal{D} is of the following form:

$$\frac{\mathcal{D}_1 :: \Pi_1; \Delta_1, x : VT_1 \vdash_{\mu_{21}} t_1 : V \wedge VT \quad \mathcal{D}_2 :: \Pi_2, \underline{x}_1 : V; \Delta_2, x_1 : VT \vdash_{\mu_{22}} t_2 : VT_2}{\Pi_1, \Pi_2; \Delta_1 \uplus \Delta_2, x : VT_1 \vdash_{\mu_{21} \otimes \mu_{22}} \mathbf{let} \underline{x}_1 \wedge x_1 = t_1 \mathbf{in} t_2 : VT_2} \text{ (ty-vlet)}$$

where we have $\Pi = \Pi_1, \Pi_2$, $\Delta = \Delta_1 \uplus \Delta_2$, $t = \mathbf{let} \underline{x}_1 \wedge x_1 = t_1 \mathbf{in} t_2$, and $\mu_2 = \mu_{21} \otimes \mu_{22}$. By induction hypothesis on \mathcal{D}_1 , we have the following derivation:

$$\mathcal{D}'_1 :: \Pi_1; \Delta_1 \vdash_{\mu_{21} \otimes \mu_{21}} t_1[x \mapsto v] : V \wedge VT$$

Thus, we have

$$\frac{\mathcal{D}'_1 \quad \mathcal{D}_2}{\Pi_1, \Pi_2; \Delta_1 \uplus \Delta_2 \vdash_{\mu_1 \otimes \mu_{21} \otimes \mu_{22}} \mathbf{let} \underline{x}_1 \wedge x_1 = t_1[x \mapsto v] \mathbf{in} t_2 : VT_2} \text{ (ty-vlet)}$$

It is clear that $t[x \mapsto v] = \mathbf{let} \underline{x}_1 \wedge x_1 = t_1[x \mapsto v] \mathbf{in} t_2$ and $\mu_1 \otimes \mu_2 = \mu_1 \otimes \mu_{21} \otimes \mu_{22}$, and we are done.

- \mathcal{D} is of the following form:

$$\frac{\mathcal{D}_1 :: \Pi_1; \Delta_1 \vdash_{\mu_{21}} t_1 : V \wedge VT \quad \mathcal{D}_2 :: \Pi_2, \underline{x}_1 : V; \Delta_2, x : VT_1, x_1 : VT \vdash_{\mu_{22}} t_2 : VT_2}{\Pi_1, \Pi_2; \Delta_1 \uplus \Delta_2, x : VT_1 \vdash_{\mu_{21} \otimes \mu_{22}} \mathbf{let} \underline{x}_1 \wedge x_1 = t_1 \mathbf{in} t_2 : VT_2} \text{ (ty-vlet)}$$

where we have $\Pi = \Pi_1, \Pi_2$, $\Delta = \Delta_1 \uplus \Delta_2$, $t = \mathbf{let} \underline{x}_1 \wedge x_1 = t_1 \mathbf{in} t_2$, and $\mu_2 = \mu_{21} \otimes \mu_{22}$. By induction hypothesis on \mathcal{D}_2 , we have the following derivation:

$$\mathcal{D}'_2 :: \Pi_2, \underline{x}_1 : V; \Delta_2, x_1 : VT \vdash_{\mu_1 \otimes \mu_{22}} t_2[x \mapsto v] : VT_2$$

Thus, we have

$$\frac{\mathcal{D}_1 \quad \mathcal{D}'_2}{\Pi_1, \Pi_2; \Delta_1 \uplus \Delta_2 \vdash_{\mu_{21} \otimes \mu_1 \otimes \mu_{22}} \mathbf{let} \underline{x}_1 \wedge x_1 = t_1 \mathbf{in} t_2[x \mapsto v] : VT_2} \text{ (ty-vlet)}$$

It is clear that $t[x \mapsto v] = \mathbf{let} \underline{x}_1 \wedge x_1 = t_1 \mathbf{in} t_2[x \mapsto v]$ and $\mu_1 \otimes \mu_2 = \mu_{21} \otimes \mu_1 \otimes \mu_{22}$, and we are done.

- \mathcal{D} is of the following form:

$$\frac{\mathcal{D}_1 :: \Pi_1; \Delta_1, x : VT_1 \vdash_{\mu_{21}} t_1 : V \wedge VT \quad \mathcal{D}_2 :: \Pi_2, \underline{x}_1 : V; \Delta_2, x : VT_1, x_1 : VT \vdash_{\mu_{22}} t_2 : VT_2}{\Pi_1, \Pi_2; \Delta_1 \uplus \Delta_2, x : VT_1 \vdash_{\mu_{21} \otimes \mu_{22}} \mathbf{let} \underline{x}_1 \wedge x_1 = t_1 \mathbf{in} t_2 : VT_2} \text{ (ty-vlet)}$$

In this case, VT_1 is a type and thus μ_1 is \square by Proposition 2.1. By induction hypothesis on \mathcal{D}_1 and \mathcal{D}_2 , we have

$$\begin{aligned} \mathcal{D}'_1 &:: \Pi_1; \Delta_1 \vdash_{\mu_{21}} t_1[x \mapsto v] : V \wedge VT \\ \mathcal{D}'_2 &:: \Pi_2, \underline{x}_1 : V; \Delta_2, x_1 : VT \vdash_{\mu_{22}} t_2[x \mapsto v] : VT_2 \end{aligned}$$

Thus, we have

$$\frac{\mathcal{D}_1 \quad \mathcal{D}'_2}{\Pi_1, \Pi_2; \Delta_1 \uplus \Delta_2 \vdash_{\mu_{21} \otimes \mu_{22}} \mathbf{let} \underline{x}_1 \wedge x_1 = t_1[x \mapsto v] \mathbf{in} t_2[x \mapsto v] : VT_2} \text{ (ty-vlet)}$$

It is clear that $t[x \mapsto v] = \mathbf{let} \underline{x}_1 \wedge x_1 = t_1[x \mapsto v] \mathbf{in} t_2[x \mapsto v]$ and $\mu_1 \otimes \mu_2 = \mu_{21} \otimes \mu_{22}$, and we are done.

All other cases can be handled similarly. ■

As usual, the soundness of the type system of λ_{view} is built on top of the following two theorems:

Theorem 2.5 (Subject Reduction) *Assume $\emptyset; (\emptyset; \emptyset) \vdash_{\mu_1} t_1 : VT$ is derivable and $\models ST_1 : \mu_1$ holds. If $(ST_1, t_1) \rightarrow_{ev/st} (ST_2, t_2)$, then $\emptyset; (\emptyset; \emptyset) \vdash_{\mu_2} t_2 : VT$ is derivable for some store type μ_2 such that $\models ST_2 : \mu_2$ holds.*

Proof By structural induction on the derivation of $\emptyset; (\emptyset; \emptyset) \vdash_{\mu_1} t_1 : VT$. ■

Theorem 2.6 (Progress) *Assume that $\emptyset; (\emptyset; \emptyset) \vdash_{\mu} t : VT$ is derivable and $\models ST : \mu$ holds. Then either t is a value or $(ST, t) \rightarrow_{ev/st} (ST', t')$ for some ST' and t' or t is of the form $E[cf(v_1, \dots, v_n)]$ such that $cf(v_1, \dots, v_n)$ is undefined.*

Proof By structural induction on the derivation \mathcal{D} of $\emptyset; (\emptyset; \emptyset) \vdash_{\mu} t : VT$. Lemma 2.2 is needed when we handling the rules **(ty-read)** and **(ty-write)**. We present an interesting case.

- \mathcal{D} is of the following form:

$$\frac{\mathcal{D}_1 :: \emptyset \vdash_{\mu_1} \underline{t} : T@L \quad \mathcal{D}_2 :: \emptyset; (\emptyset; \emptyset) \vdash t_1 : \mathbf{ptr}(L)}{\emptyset; (\emptyset; \emptyset) \vdash_{\mu_1 \otimes \mu_2} \mathbf{read}(\underline{t}, t_1) : (T@L) \wedge T} \quad \mathbf{(ty-read)}$$

where $t = \mathbf{read}(\underline{t}, t_1)$ and $\mu = \mu_1 \otimes \mu_2$. If t_1 is not a value, then the case easily follows from the induction hypothesis on \mathcal{D}_2 . We now assume that t_1 is a value. Hence, $t_1 = l$ for some constant address l and $L = l$. Since $ST \models \mu$ holds, we have $ST = ST_1 \otimes ST_2$ such that both $ST_1 \models \mu_1$ and $ST_2 \models \mu_2$ hold. By Lemma 2.2, we have $ST_1 \models T@L$ and thus $ST_1 = [l \mapsto v]$ for some v such that $\emptyset; (\emptyset; \emptyset) \vdash v : T$ holds. Clearly, we have $(ST, t) \rightarrow_{ev/st} (ST', t')$ for $ST' = ST$ and $t' = v$.

All other cases can be handled similarly. ■

By Theorem 2.5 and Theorem 2.6, we can readily infer that if $\emptyset; (\emptyset; \emptyset) \vdash_{\mu} t : VT$ is derivable and $\models ST : \mu$ holds, then either the evaluation of (ST, t) reaches (ST', v) for some state ST' and value v or it continues forever.

Clearly, We can define a function $|\cdot|$ that erases all the proof terms in a given dynamic term. For instance, some key cases in the definition of the erasure function is given as follows:

$$\begin{aligned} |\mathbf{let } \underline{x} \wedge x = t_1 \mathbf{in } t_2| &= \mathbf{let } x = |t_1| \mathbf{in } t_2 \\ |\underline{t} \wedge t| &= |t| \\ |\lambda \underline{x}. v| &= |v| \\ |t(\underline{t})| &= |t| \\ |\mathbf{read}(\underline{t}, t)| &= \mathbf{read}(|t|) \\ |\mathbf{write}(\underline{t}, t_1, t_2)| &= \mathbf{write}(|t_1|, |t_2|) \end{aligned}$$

It is then straightforward to show that a dynamic term evaluates to a value if and only if the erasure of the dynamic term evaluates to the erasure of the value. Thus, there is no need to keep proof terms at run-time: They are only needed for the purpose of type-checking.

sorts	$\sigma ::= \text{addr} \mid \text{bool} \mid \text{int} \mid \text{view} \mid \text{type} \mid \text{viewtype}$
static contexts	$\sigma ::= \emptyset \mid \Sigma, a : \sigma$
static addr.	$L ::= a \mid l \mid L + I$
static int.	$I ::= a \mid i \mid c_I(s_1, \dots, s_n)$
static prop.	$P ::= a \mid b \mid c_P(s_1, \dots, s_n) \mid \neg P \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \supset P_2$
views	$V ::= \dots \mid P \supset V \mid \forall a : \sigma. V \mid P \wedge V \mid \exists a : \sigma. V$
types	$T ::= \dots \mid \mathbf{bool}(P) \mid \mathbf{int}(I) \mid P \supset T \mid \forall a : \sigma. T \mid P \wedge T \mid \exists a : \sigma. T$
viewtypes	$VT ::= \dots \mid P \supset VT \mid \forall a : \sigma. VT \mid P \wedge VT \mid \exists a : \sigma. VT$

 Figure 5: The syntax for the statics of $\lambda_{view}^{\forall, \exists}$

3 Extension

While it supports both views and viewtypes, λ_{view} is essentially based on the simply typed language calculus. This makes it difficult to truly reap the benefits of views and viewtypes. In this section, we outline an extension from λ_{view} to $\lambda_{view}^{\forall, \exists}$ to include universally quantified types as well as existentially quantified types, greatly facilitating the use of views and viewtypes in programming.

Like an applied type system (Xi, 2004; Xi, 2003), $\lambda_{view}^{\forall, \exists}$ consists of a static component (statics) and a dynamic component (dynamics). The syntax for the statics of $\lambda_{view}^{\forall, \exists}$ is given in Figure 5. The statics itself is a simply typed language and a type in it is called a *sort*. We assume the existence of the following basic sorts in $\lambda_{view}^{\forall, \exists}$: *addr*, *bool*, *int*, *type*, *view* and *viewtype*; *addr* is the sort for addresses, and *bool* is the sort for boolean constants, and *int* is the sort for integers, and *type* is the sort for types, and *view* is the sort for views, and *viewtype* is the sort for viewtypes. We use a for static variables, l for address constants $\mathbf{I}_0, \mathbf{I}_1, \dots$, b for boolean values *true* and *false*, and i for integers $0, -1, 1, \dots$. We may also use $\mathbf{0}$ for the null address \mathbf{I}_0 . A term s in the statics is called a static term, and we use $\Sigma \vdash s : \sigma$ to mean that s can be assigned the sort σ under Σ . The rules for assigning sorts to static terms are all omitted as they are completely standard.

We may also use L, P, I, T, V, VT for static terms of sorts *addr*, *bool*, *int*, *type*, *view*, and *viewtype*, respectively. We assume some primitive functions c_I when forming static terms of sort *int*; for instance, we can form terms such as $I_1 + I_2$, $I_1 - I_2$, $I_1 * I_2$ and I_1 / I_2 . Also we assume certain primitive functions c_P when forming static terms of sort *bool*; for instance, we can form propositions such as $I_1 \leq I_2$ and $I_1 \geq I_2$, and for each sort σ we can form a proposition $s_1 =_\sigma s_2$ if s_1 and s_2 are static terms of sort σ ; we may omit the subscript σ in $=_\sigma$ if it can be readily inferred from the context. In addition, given L and I , we can form an address $L + I$, which equals \mathbf{I}_{n+i} if $L = \mathbf{I}_n$ and $I = i$ and $n + i \geq 0$.

We use \overline{P} for a sequence of propositions and $\Sigma; \overline{P} \models P$ for a constraint that means for any $\Theta : \Sigma$, if each proposition in $\overline{P}[\Theta]$ holds then so does $P[\Theta]$.

In addition, we introduce two type constructors **bool** and **int**; given a proposition P , **bool**(P) is the singleton type in which the only value is the truth value of P ; similarly, given an integer I ,

$\mathbf{int}(I)$ is the singleton type in which the only value is the integer I . Obviously, the previous types \mathbf{Bool} and \mathbf{Int} can now be defined as $\exists a : \mathbf{bool}.\mathbf{bool}(a)$ and $\exists a : \mathbf{int}.\mathbf{int}(a)$, respectively.

proof terms	\underline{t}	$::=$	$\dots \mid \supset^+(\underline{t}) \mid \supset^-(\underline{t}) \mid \forall^+(\underline{t}) \mid \forall^-(\underline{t}) \mid$ $\wedge(\underline{t}) \mid \mathbf{let} \wedge(\underline{x}) = \underline{t}_1 \mathbf{in} \underline{t}_2 \mid \exists(\underline{t}) \mid \mathbf{let} \exists(\underline{x}) = \underline{t}_1 \mathbf{in} \underline{t}_2$
dynamic terms	t	$::=$	$\dots \mid \supset^+(v) \mid \supset^-(t) \mid \forall^+(v) \mid \forall^-(t) \mid$ $\wedge(t) \mid \mathbf{let} \wedge(\underline{x}) = \underline{t}_1 \mathbf{in} \underline{t}_2 \mid \exists(t) \mid \mathbf{let} \exists(\underline{x}) = \underline{t}_1 \mathbf{in} \underline{t}_2$
values	v	$::=$	$\dots \mid \supset^+(v) \mid \forall^+(v) \mid \wedge(v) \mid \exists(v)$

Figure 6: The syntax for the dynamics of $\lambda_{view}^{\forall, \exists}$

Some (additional) syntax for the dynamics of $\lambda_{view}^{\forall, \exists}$ is given in Figure 6. The markers $\supset^+(\cdot)$, $\supset^-(\cdot)$, $\forall^+(\cdot)$, $\forall^-(\cdot)$, $\wedge(\cdot)$ and $\exists(\cdot)$ are primarily introduced to prove the soundness of the type system of $\lambda_{view}^{\forall, \exists}$, and please see (Xi, 2004; Xi, 2003) for explanation.

We can now also introduce (built-in) memory allocation/deallocation functions $alloc$ and $free$ and assign them the following constant types:

$$\begin{aligned}
 alloc & : \quad \forall l.l \geq 0 \supset (\mathbf{int}(l) \Rightarrow \exists \lambda.\lambda \neq \mathbf{0} \wedge (\mathbf{arrayView}(\mathbf{1}, l, \lambda) \wedge \mathbf{ptr}(\lambda))) \\
 free & : \quad \forall \tau.\forall l.l \geq 0 \supset (\mathbf{arrayView}(\tau, l, \lambda) \wedge (\mathbf{ptr}(\lambda) * \mathbf{int}(l)) \Rightarrow \mathbf{1})
 \end{aligned}$$

When applied to a natural number n , $alloc$ returns a pointer (that is not null) pointing to a newly allocated array of n units; when applied to a pointer pointing to an array of size n , $free$ frees the array. Note that how these two functions are implemented is inessential here as long as the implementations meet the constant types.

A judgment for assigning a view to a proof is now of the form $\Sigma; \overline{P}; \Pi \vdash_{\mu} \underline{t} : V$, and the rules in Figure 3 need to be modified properly. Intuitively, such a judgment means that $\Pi[\Theta] \vdash_{\mu} \underline{t} : V[\Theta]$ holds for any substitution $\Theta : \Sigma$ such that each P in $\overline{P}[\Theta]$ holds. Some additional rules for assigning views to proof terms are given in Figure 7.

Similarly, a judgment for assigning a viewtype to a dynamic term is now of the form $\Sigma; \overline{P}; \Pi; \Delta \vdash t : VT$, and the rules in Figure 4 need to be modified properly. Some additional rules for assigning viewtypes to dynamic terms are given in Figure 8.

Following the development as is detailed in (Xi, 2004), we expect it to be a standard routine to establish the soundness of the type system of $\lambda_{view}^{\forall, \exists}$. The challenge here is really not in the proof of the soundness; it is instead in the formulation of the rules presented in Figure 3, Figure 7, Figure 4 and Figure 8 for assigning views and viewtypes to proof terms and dynamic terms, respectively.

4 Examples

The type system of $\lambda_{view}^{\forall, \exists}$ is considerably involved, and one may suspect that it is only of theoretical interest. This, however, is not the case. We have actually designed a programming language ATS with a type system based on that of $\lambda_{view}^{\forall, \exists}$, and a prototype implementation of ATS is already

$$\begin{array}{c}
 \frac{\Sigma; \overline{P}, P; \Pi \vdash_{\mu} \underline{t} : V}{\Sigma; \overline{P}; \Pi \vdash_{\mu} \supset^+(\underline{t}) : P \supset V} \text{ (vw-}\supset^+\text{)} \\
 \frac{\Sigma; \overline{P}; \Pi \vdash_{\mu} \underline{t} : P \supset V \quad \Sigma; \overline{P} \models P}{\Sigma; \overline{P}; \Pi \vdash_{\mu} \supset^-(\underline{t}) : V} \text{ (vw-}\supset^-\text{)} \\
 \frac{\Sigma, a : \sigma; \overline{P}; \Pi \vdash_{\mu} \underline{t} : V}{\Sigma; \overline{P}; \Pi \vdash_{\mu} \forall^+(\underline{t}) : \forall a : \sigma. V} \text{ (vw-}\forall^+\text{)} \\
 \frac{\Sigma; \overline{P}; \Pi \vdash_{\mu} \underline{t} : V \quad \Sigma \vdash s : \sigma}{\Sigma; \overline{P}; \Pi \vdash_{\mu} \forall^-(\underline{t}) : V[a \mapsto s]} \text{ (vw-}\forall^-\text{)} \\
 \frac{\Sigma; \overline{P} \models P \quad \Sigma; \overline{P}; \Pi \vdash_{\mu} \underline{t} : V}{\Sigma; \overline{P}; \Pi \vdash_{\mu} \wedge(\underline{t}) : P \wedge V} \text{ (vw-}\wedge^+\text{)} \\
 \frac{\Sigma; \overline{P}; \Pi_1 \vdash_{\mu_1} \underline{t}_1 : P \wedge V_1 \quad \Sigma; \overline{P}, P; \Pi_2, \underline{x} : V_1 \vdash_{\mu_2} \underline{t}_2 : V_2}{\Sigma; \overline{P}; \Pi_1, \Pi_2 \vdash_{\mu_1 \otimes \mu_2} \mathbf{let} \wedge(\underline{x}) = \underline{t}_1 \mathbf{in} \underline{t}_2 : V_2} \text{ (vw-}\wedge^-\text{)} \\
 \frac{\Sigma \vdash s : \sigma \quad \Sigma; \overline{P}; \Pi \vdash_{\mu} \underline{t} : V[a \mapsto s]}{\Sigma; \overline{P}; \Pi \vdash_{\mu} \exists(\underline{t}) : \exists a : \sigma. V} \text{ (vw-}\exists^+\text{)} \\
 \frac{\Sigma; \overline{P}; \Pi_1 \vdash_{\mu_1} \underline{t}_1 : \exists a : \sigma. V_1 \quad \Sigma, a : \sigma; \overline{P}; \Pi_2, \underline{x} : V_1 \vdash_{\mu_2} \underline{t}_2 : V_2}{\Sigma; \overline{P}; \Pi_1, \Pi_2 \vdash_{\mu_1 \otimes \mu_2} \mathbf{let} \exists(\underline{x}) = \underline{t}_1 \mathbf{in} \underline{t}_2 : V_2} \text{ (vw-}\exists^-\text{)}
 \end{array}$$

Figure 7: Some additional rules for assigning views to proof terms

accessible on-line (Xi, 2003). In this section, we present some simple examples taken from our current implementation, aiming at providing the reader with some concrete feel as to how views and viewtypes can allow for the construction of memory-safe programs while supporting explicit pointer manipulation.

A simple function written in ATS is given in Figure 9 that swaps the contents stored at two (distinct) addresses. The syntax of ATS bears a great deal of resemblance to that of Standard ML (Milner et al., 1997). The header of the function indicates that the following type is assigned to *swap*:

$$\forall \tau_1. \forall \tau_2. \forall \lambda_1. \forall \lambda_2. (\tau_1 @ \lambda_1 \otimes \tau_2 @ \lambda_2) \wedge (\mathbf{ptr}(\lambda_1) * \mathbf{ptr}(\lambda_2)) \rightarrow (\tau_2 @ \lambda_1 \otimes \tau_1 @ \lambda_2) \wedge \mathbf{1}$$

The type of *swap* means that (1) *swap* can be called on two pointers L_1 and L_2 only if two values of some types T_1 and T_2 are stored at addresses L_1 and L_2 , respectively, and (2) two values of types T_2 and T_1 are stored at addresses L_1 and L_2 , respectively, when the call returns.

In the concrete syntax in Figure 9, we use $'(\dots)$ to form tuples, where the quote symbol ($'$) is solely for the purpose of parsing. For instance, $'()$ stands for the unit (i.e., the tuple of length 0). Also, the bar symbol ($|$) is used as a separator (like the comma symbol $,$). It should be

$$\begin{array}{c}
 \frac{\Sigma; \overline{P}; \Pi_1; \Delta_1 \vdash_{\mu_1} t_1 : \mathbf{bool}(P) \quad \Sigma; \overline{P}, P; \Pi_2; \Delta_2 \vdash_{\mu_2} t_2 : VT \quad \Sigma; \overline{P}, \neg P; \Pi_2; \Delta_2 \vdash_{\mu_2} t_3 : VT}{\Sigma; \overline{P}; \Pi_1, \Pi_2; \Delta_1 \uplus \Delta_2 \vdash_{\mu_1 \otimes \mu_2} \mathbf{if}(t_1, t_2, t_3) : VT} \text{ (ty-if)} \\
 \frac{\Sigma; \overline{P}, P; \Pi; \Delta \vdash_{\mu} t : VT}{\Sigma; \overline{P}; \Pi; \Delta \vdash_{\mu} \supset^+(t) : P \supset VT} \text{ (ty-}\supset^+\text{)} \\
 \frac{\Sigma; \overline{P}; \Pi; \Delta \vdash_{\mu} t : P \supset VT \quad \Sigma; \overline{P} \models P}{\Sigma; \overline{P}; \Pi; \Delta \vdash_{\mu} \supset^-(t) : VT} \text{ (ty-}\supset^-\text{)} \\
 \frac{\Sigma, a : \sigma; \overline{P}; \Pi; \Delta \vdash_{\mu} t : VT}{\Sigma; \overline{P}; \Pi; \Delta \vdash_{\mu} \forall^+(t) : \forall a : \sigma. VT} \text{ (ty-}\forall^+\text{)} \\
 \frac{\Sigma; \overline{P}; \Pi; \Delta \vdash_{\mu} t : \forall a : \sigma. VT \quad \Sigma \vdash s : \sigma}{\Sigma; \overline{P}; \Pi; \Delta \vdash_{\mu} \forall^-(t) : VT[a \mapsto s]} \text{ (ty-}\forall^-\text{)} \\
 \frac{\Sigma; \overline{P} \models P \quad \Sigma; \overline{P}; \Pi; \Delta \vdash_{\mu} t : VT}{\Sigma; \overline{P}; \Pi; \Delta \vdash_{\mu} \wedge(t) : P \wedge VT} \text{ (ty-}\wedge^+\text{)} \\
 \frac{\Sigma; \overline{P}; \Pi_1; \Delta_1 \vdash_{\mu_1} t_1 : P \wedge VT_1 \quad \Sigma; \overline{P}, P; \Pi_2; \Delta_2, x : VT_1 \vdash_{\mu_2} t_2 : VT_2}{\Sigma; \overline{P}; \Pi_1, \Pi_2; \Delta_1 \uplus \Delta_2 \vdash_{\mu_1 \otimes \mu_2} \mathbf{let} \ \wedge(x) = t_1 \ \mathbf{in} \ t_2 : VT_2} \text{ (ty-}\wedge^-\text{)} \\
 \frac{\Sigma \vdash_{\mu} s : \sigma \quad \Sigma; \overline{P}; \Pi; \Delta \vdash_{\mu} t : VT[a \mapsto s]}{\Sigma; \overline{P}; \Pi; \Delta \vdash_{\mu} \exists(t) : \exists a : \sigma. VT} \text{ (ty-}\exists^+\text{)} \\
 \frac{\Sigma; \overline{P}; \Pi_1; \Delta_1 \vdash_{\mu_1} t_1 : \exists a : \sigma. VT_1 \quad \Sigma, a : \sigma; \overline{P}; \Pi_2; \Delta_2, x : VT_1 \vdash_{\mu_2} t_2 : VT_2}{\Sigma; \overline{P}; \Pi_1, \Pi_2; \Delta_1 \uplus \Delta_2 \vdash_{\mu_1 \otimes \mu_2} \mathbf{let} \ \exists(x) = t_1 \ \mathbf{in} \ t_2 : VT_2} \text{ (ty-}\exists^-\text{)}
 \end{array}$$

Figure 8: Some additional rules for assigning viewtypes to proof terms

straightforward to related the concrete syntax to the formal syntax of $\lambda_{view}^{\forall, \exists}$ (assuming that the reader is familiar with the SML syntax).

A clearly noticeable weakness in many typed programming languages lies in the treatment of the allocation and initialization of arrays. For instance, the allocation and initialization of an array in SML is atomic and cannot be done separately. Therefore, copying an array requires a new array be allocated and then initialized before copying can actually proceed. Though the initialization of the newly allocated array is completely useless, it unfortunately cannot be avoided. In $\lambda_{view}^{\forall, \exists}$ (extended with recursive stateful views), a function of the following type can be readily implemented that replaces elements of type T_1 in an array with elements of type T_2 when a function of type $T_1 \rightarrow T_2$ is given:

$$\begin{array}{l}
 \forall \tau_1. \forall \tau_2. \forall \iota. \forall \lambda. \iota \geq 0 \supset \\
 (\mathbf{arrayView}(\tau_1, n, \lambda) \wedge (\mathbf{ptr}(\lambda) * \mathbf{int}(\iota) * (\tau_1 \rightarrow \tau_2)) \rightarrow \\
 \mathbf{arrayView}(\tau_2, n, \lambda) \wedge \mathbf{ptr}(\lambda))
 \end{array}$$

```

fun swap {a1:type, a2:type, l1:addr, l2: addr}
  (pf1: a1 @ l1, pf2: a2 @ l2 | p1: ptr l1, p2: ptr l2)
  : (a1 @ l2, a2 @ l1 | unit) =
  let
    val '(pf1' | v1) = read (pf1 | p1)
    val '(pf2' | v2) = read (pf2 | p2)
    val '(pf1'' | _) = write (pf1' | p1, v2)
    val '(pf2'' | _) = write (pf2' | p2, v1)
  in
    '(pf1'', pf2'' | '())
  end

```

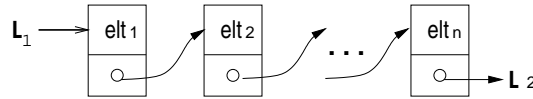
Figure 9: A simple swap function

With such a function, the allocation and initialization of an array can clearly be separated. In Figure 10, we present an implementation of in-place array map function in ATS. Note that *arrayView* is declared as a recursive stateful view constructor in Figure 1.¹ Note that for a proof *pf* of view *arrayView*(*T*, *I*, *L*) for some type *T*, integer *I* > 0 and address *L*, the following syntax in Figure 10 means that *pf* is decomposed into two proofs *pf*₁ and *pf*₂ of views *T*@*L* and *arrayView*(*T*, *I* - 1, *L* + 1), respectively:

```
prval ArraySome (pf1, pf2) = pf
```

The rest of the syntax in Figure 10 should then be easily accessible.

The last example we present is in Figure 11, where a recursive view constructor *slseg* is declared. Note that we write $(T_0, \dots, T_n)@L$ for a sequence of views: $T_0@(L+0), \dots, T_n@(L+n)$. Given a type *T*, an integer *I*, and two addresses *L*₁ and *L*₂, *slseg*(*T*, *I*, *L*₁, *L*₂) is a view for a singly-linked list segment pictured as follows:



such that

- each element of the segment is of type *T*, and
- the length of the segment is *n*, and
- the segment starts at *L*₁ and ends at *L*₂.

There are two view proof constructors *SlsegNone* and *SlsegSome* associated with *slseg*. A singly-linked list is just a special case of singly-linked list segment that ends at the null address. Therefore, *sllist*(*T*, *I*, *L*) is a view for a singly-linked list pictured as follows:

¹Though the notion of recursive stateful view is not present in $\lambda_{view}^{\forall, \exists}$, it should be understood that this notion can be readily incorporated.

```

fun arrayMap {a1: type, a2: type, n: int, l: addr | n >= 0}
  (pf: arrayView (a1, n, l) | A: ptr l, n: int n, f: a1 -> a2)
  : '(arrayView (a2, n, l) | unit) =
  if n > 0 then
    let
      prval ArraySome (pf1, pf2) = pf
      val '(pf1' | v) = read (pf1 | A)
      val '(pf1'' | _) = write (pf1' | A, f v)
      val '(pf2' | _) = arrayMap (pf2 | A + 1, n - 1, f)
    in
      '(ArraySome (pf1'', pf2') | '())
    end
  else
    let prval ArrayNone = pf in '(ArrayNone | '()) end

```

Figure 10: An implementation of in-place array map function

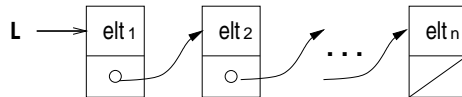
```

dataview slseg (type, int, addr, addr) =
  | {a:type, l:addr} SlsegNone (a, 0, l, l)
  | {a:type, n:int, first, next, last | n >= 0, first <> null}
  // 'first <> null' is added so that nullity test can
  // be used to check whether a list segment is empty.
  SlsegSome (a, n+1, first, last) of
    ((a, ptr next) @ first, slseg (a, n, next, last))

viewdef sllist (a, n, l) = slseg (a, n, l, null)

```

Figure 11: A dataview for singly linked list segments



such that each element in it is of type T and its length is I . A in-place reversal function on linked-lists can be found in (Xi et al., 2004).

5 Related Work and Conclusion

The issue of memory safety is of great importance in programming, and we have seen a large body of research (e.g. (Wadler, 1990; Chirimar et al., 1996; Turner and Wadler, 1999; Kobayashi, 1999; Igarashi and Kobayashi, 2000; Hofmann, 2000)) applying type theory based on linear logic (Girard, 1987) to memory management. Noticeably, the work in (Petersen et al., 2003) that attempts

to give an account for data layout based on ordered linear logic (Polakow and Pfenning, 1999) is closely related to λ_{view} in the aspect that memory allocation and data initialization are allowed to be separated completely, though it is yet to see how recursive data structures such as arrays and linked lists can be properly handled with ordered linear logic. On the other hand, these data structures can be readily dealt with through recursive stateful views. However, we are currently unable to characterize in precise terms the expressiveness of recursive stateful views in modeling memory layouts.

Along a related but different line of research, separation logic (Reynolds, 2002) has recently been introduced as an extension to Hoare logic in support of reasoning on mutable data structures. The effectiveness of separation logic in establishing program correctness is convincingly demonstrated in various nontrivial examples (e.g., singly-linked lists and doubly-linked lists). In a (rather) broad sense, $\lambda_{view}^{\forall, \exists}$ can be viewed as a novel attempt to combine type theory with (a form of) separation logic. In particular, the treatment of functions as first-class values is a significant part of $\lambda_{view}^{\forall, \exists}$, which is not addressed in separation logic.

There have been a large number of research activities on verifying program safety properties by tracking state changes. For instance, Cyclone (Jim et al., 2001) allows the programmer to specify safe stack and region memory allocation; both CQual (Foster et al., 2002) and Vault (Fahndrich and Deline, 2002) support some form of resource usage protocol verification; ESC (Detlefs, 1996) enables the programmer to state various kinds of program invariants and then employs theorem proving to prove them; in addition, CCured (Necula et al., 2002) uses program analysis to show the safety of mostly unannotated C programs. In this paper, we are primarily interested in providing a framework based on type theory to reason about program states. This aspect is also shared in the research on an effective theory of type refinements (Mandelbaum et al., 2003), where the aim is to develop a general theory of type refinements for reasoning about program states. However, the notion of viewtype seems to have no direct counterparts there.

References

- Chirimar, J., Gunter, C. A., and Riecke, G. (1996). Reference Counting as a Computational Interpretation of Linear Logic. *Journal of Functional Programming*, 6(2):195–244.
- Detlefs, D. (1996). An overview of the extended static checking system. In *Workshop on Formal Methods in Software Practice*.
- Fahndrich, M. and Deline, R. (2002). Adoption and Focus: Practical Linear Types for Imperative Programming. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 13–24, Berlin.
- Foster, J., Terauchi, T., and Aiken, A. (2002). Flow-sensitive Type Qualifiers. In *ACM Conference on Programming Language Design and Implementation*, pages 1–12, Berlin.
- Girard, J.-Y. (1987). Linear logic. *Theoretical Computer Science*, 50(1):1–101.
- Hofmann, M. (2000). A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289.

- Igarashi, A. and Kobayashi, N. (2000). Garbage Collection Based on a Linear Type System. In *Preliminary Proceedings of the 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC '00)*.
- Jim, T., Morrisett, G., Grossman, D., Hicks, M., Baudet, M., Harris, M., and Wang, Y. (2001). Cyclone, a Safe Dialect of C. Available at <http://www.cs.cornell.edu/Projects/cyclone/>.
- Kobayashi, N. (1999). Quasi-linear types. In *Proceedings of the 26th ACM Sigplan Symposium on Principles of Programming Languages (POPL '99)*, pages 29–42, San Antonio, Texas, USA.
- Mandelbaum, Y., Walker, D., and Harper, R. (2003). An effective theory of type refinements. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 213–226, Uppsala, Sweden.
- Milner, R., Tofte, M., Harper, R. W., and MacQueen, D. (1997). *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts.
- Necula, G. C., McPeak, S., and Weimer, W. (2002). CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 128–139, London.
- Petersen, L., Harper, R., Crary, K., and Pfenning, F. (2003). A Type Theory for Memory Allocation and Data Layout. In *Proceedings of the 30th ACM Sigplan Symposium on Principles of Programming Languages (POPL '03)*, pages 172–184, New Orleans, Louisiana, USA.
- Polakow, J. and Pfenning, F. (1999). Relating natural deduction and sequent calculus for intuitionistic non-commutative linear logic. In Scedrov, A. and Jung, A., editors, *Proceedings of the 15th Conference on Mathematical Foundations of Programming Semantics*, New Orleans, Louisiana. Electronic Notes in Theoretical Computer Science, Volume 20.
- Reynolds, J. (2002). Separation Logic: a logic for shared mutable data structures. In *Proceedings of 17th IEEE Symposium on Logic in Computer Science (LICS '02)*.
- Turner, D. N. and Wadler, P. (1999). Operational interpretations of linear logic. *Theoretical Computer Science*, 227(1–2):231–248.
- Wadler, P. (1990). Linear types can change the world. In *TC 2 Working Conference on Programming Concepts and Methods (Preprint)*, pages 546–566, Sea of Galilee.
- Xi, H. (2003). Applied Type System. Available at: <http://www.cs.bu.edu/~hwxi/ATS>.
- Xi, H. (2004). Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003*, pages 394–408. Springer-Verlag LNCS 3085.
- Xi, H. and Zhu, D. (2004). Views, Types and Viewtypes. Available at: <http://www.cs.bu.edu/~hwxi/ATS/PAPER/VsTsVTs.ps>.
- Xi, H., Zhu, D., and Li, Y. (2004). Applied Type System with Stateful Views. Available at: <http://www.cs.bu.edu/~hwxi/ATS/PAPER/ATSSV.ps>.